

THE APPLICATION OF ANNA AND FORMAL METHODS  
AS AN ADA PROGRAM DESIGN LANGUAGE

By

C. M. BYRNES

OCTOBER 1986

Prepared for  
DEPUTY COMMANDER FOR DEVELOPMENT PLANS AND SUPPORT SYSTEMS  
ELECTRONIC SYSTEMS DIVISION  
AIR FORCE SYSTEMS COMMAND  
UNITED STATES AIR FORCE

Hanscom Air Force Base, Massachusetts



Approved for public release;  
distribution unlimited.

Project No. 5720  
Prepared by  
THE MITRE CORPORATION  
Bedford, Massachusetts  
Contract No. F19628-86-C-0001

ADA175120

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

### REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.



---

JUNE I. R. BABSON, Major, USAF  
Chief, Computer Technology,  
Operations & Support Division

FOR THE COMMANDER



---

ROBERT J. KENT  
Director, Software Design Center  
Deputy Commander for Development  
Plans and Support Systems

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE																
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS														
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.														
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE																
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MTR-10067 ESD-TR-86-276		5. MONITORING ORGANIZATION REPORT NUMBER(S)														
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation		6b. OFFICE SYMBOL (If applicable)		7a. NAME OF MONITORING ORGANIZATION												
6c. ADDRESS (City, State and ZIP Code) Burlington Road Bedford, MA 01730		7b. ADDRESS (City, State and ZIP Code)														
8a. NAME OF FUNGION/SPONSORING ORGANIZATION Deputy Commander for (cont)		8b. OFFICE SYMBOL (If applicable) XRS		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-86-C-0001												
8c. ADDRESS (City, State and ZIP Code) Electronic Systems Division, AFSC Hanscom AFB, MA 01731-5000		10. SOURCE OF FUNGION NOS. <table border="1"><thead><tr><th>PROGRAM ELEMENT NO.</th><th>PROJECT NO.</th><th>TASK NO.</th><th>WORK UNIT NO.</th></tr></thead><tbody><tr><td></td><td>5720</td><td></td><td></td></tr></tbody></table>			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.		5720						
PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.													
	5720															
11. TITLE (Include Security Classification) THE APPLICATION OF ANNA AND FORMAL (cont)																
12. PERSONAL AUTHOR(S) Byrnes, C. M.																
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) 1986 October												
15. PAGE COUNT 52																
16. SUPPLEMENTARY NOTATION																
17. COSATI CODES <table border="1"><thead><tr><th>FIELD</th><th>GROUP</th><th>SUB. GR.</th></tr></thead><tbody><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></tbody></table>		FIELD	GROUP	SUB. GR.										18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Ada PDL CAIS ANNA DOD-STD-2167 Buhr Design Methodology Formal Methods		
FIELD	GROUP	SUB. GR.														
19. ABSTRACT (Continue on reverse if necessary and identify by block number) As part of its support for the introduction of Ada* technology into Air Force projects, the MITRE Software Center began an investigation into how Ada could be used as a Program Design Language (PDL) within the framework of the software development process called for in DOD-STD-2167 and its associated Data Item Descriptions (DIDs). This investigation also looked into the use of formal software development methods within an Ada PDL. The investigation took the form of a model design exercise called the Micro Interactive Monitor System (MIMSY). Project documentation and two preliminary designs were produced. These designs used a combination of Buhr's diagram notations and Luckham's ANNotated Ada (ANNA) language and Task Sequencing Language (TSL) as the program design languages. This report summarizes the work performed and the lessons learned about Ada PDLs, ANNA, TSL and formal methods.  *Ada is a trademark of the Ada Joint Program Office.																
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> OTIC USERS <input type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION Unclassified														
22a. NAME OF RESPONSIBLE INDIVIDUAL Diana F. Arimento		22b. TELEPHONE NUMBER (Include Area Code) (617)271-7454		22c. OFFICE SYMBOL Mail Stop D230												

DD FORM 1473, 83 APR

EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

- 8a. Development Plans and Support Systems
- 11. METHODS AS AN ADA PROGRAM DESIGN LANGUAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

## ACKNOWLEDGMENTS

This document has been prepared by The MITRE Corporation under Project No. 5720, Contract No. F19628-86-C-0001, ESD/MITRE Software Center General Support. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts 01731.

The author thanks everyone at MITRE who worked on the MIMSY project; especially Marlene Hazle, Steve Litvintchouk, Shabbir Dahod, Maryellen Costello, Diana Parton, John Maurer, and Mark Gerhardt.

## TABLE OF CONTENTS

<u>Section</u>	<u>Page</u>
1 INTRODUCTION	1
1.1 RATIONALE FOR THIS INVESTIGATION	1
1.2 APPROACH TAKEN	1
2 MIMSY'S FUNCTIONALITY	5
2.1 OPERATIONS TO BE PERFORMED	5
2.2 MAJOR MIMSY COMPONENTS	5
3 DESIGN METHODOLOGY USED WITH MIMSY	7
3.1 DESCRIPTION OF DESIGN METHODOLOGY	7
3.2 DESCRIPTION OF DESIGN NOTATION	8
3.3 RATIONALE FOR USING THIS DESIGN METHODOLOGY	9
4 RELATIONSHIP OF ANNA TO FORMAL METHODS	13
4.1 ANNA IMPROVES ADA'S COVERAGE OF FORMAL METHODS	13
4.2 ANNA IS WELL-INTEGRATED INTO ADA	14
4.3 TSL COVERS ADA TASKING	15
4.4 ANNA USABLE WITH DIFFERENT APPROACHES	16
4.5 AN ANNA EXAMPLE FROM MIMSY	17
4.6 ANNA TOOLS	17
4.7 TSL TOOLS	18
5 MIMSY'S USE OF FORMAL METHODS	19
5.1 BUHR DIAGRAM'S RELATIONSHIP TO FORMAL METHODS	19
5.2 MIMSY USED ALGEBRAIC SPECIFICATION APPROACH	20

## TABLE OF CONTENTS (Concluded)

<u>Section</u>	<u>Page</u>
5.3 MIMSY USED ANNA AND TSL AS THE PDL	21
5.4 USE OF ANNA WITH CAISETTE PACKAGE	23
5.5 ADVANTAGES OF USING FORMAL METHODS IN MIMSY	24
5.6 REVIEW PROCESS WHEN USING FORMAL METHODS	26
6 CONCLUSIONS	27
APPENDIX A ANNA EXAMPLE	29
APPENDIX B CAISETTE EXAMPLE	33
LIST OF REFERENCES	43

## LIST OF ILLUSTRATIONS

<u>Figure</u>		<u>Page</u>
1	ANNA Example: Preconditions and Postconditions	30
2	ANNA Example: Package Axioms	31
3	CAISette Without ANNA or TSL (OPEN and CLOSE)	34
4	CAISette Without ANNA or TSL (PUT)	35
5	CAISette Example With ANNA (OPEN)	36
6	CAISette Example With ANNA (CLOSE)	37
7	CAISette Example With ANNA (SET_POSITION)	38
8	CAISette Example With ANNA (PUT)	39
9	CAISette Example With TSL (OPEN and CLOSE)	40
10	CAISette Example With TSL (PUT)	41

## SECTION 1

### INTRODUCTION

#### 1.1 RATIONALE FOR THIS INVESTIGATION

The growing interest in using Ada as a Program Design Language (PDL) as well as an implementation language has led to several different proposals for how to use Ada as a Design Language. More and more projects require the use of Ada as a PDL in an attempt to standardize on a common PDL and to ease the transition to Ada as the implementation (programming) language. Ada is much more rigorously defined than most PDLs, and Ada has a much richer set of constructs. Using Ada as a PDL will represent a major change to most current software designers.

Recently the government software development and documentation standards have changed with the introduction of DOD-STD-2167 [DOD85] and the revisions to MIL-STD-483A, MIL-STD-490A and MIL-STD-1521B. With the use of Ada as a PDL likely in some of the documents called for in DOD-STD-2167 and its associated DIDs, project managers need some guidance about how Ada as a PDL relates to the other design documentation. Examples of the use of Ada as a PDL and a documentation "boilerplate" will be useful to current and future projects.

There is a growing awareness in the software development field of the importance of formal mathematical methods in producing accurate and reliable software. Many of the advanced features in Ada (such as packages, strong typing and overloading) were included in the language to help support some formal methods. While the Ada language (as defined in MIL-STD-1815A) contains support for some formal methods, there are other formal methods that are not directly supported by the syntax and semantics of Ada. Several recent Ada PDL proposals have tried to improve Ada's support of these missing formal methods through extensions to the language and through support tools [Inte84]. As the need for highly accurate and reliable software grows, the need for Ada PDLs that support many types of formal methods will grow also.

#### 1.2 APPROACH TAKEN

The MITRE Software Center decided to investigate the use of Ada as a Program Design Language (PDL), including formal methods and within a DOD-STD-2167 framework, by performing a small design exercise. The problem that was chosen was a subset of the functionality

of the APSE Interactive Monitor (AIM) [TI83]. Because of a limit on the personnel, resources and time that were available, only a very minimal subset of the AIM functionality was implemented. Our system was called the Micro Interactive Monitor SYstem (MIMSY).

To enhance the realism of the design exercise, all the Software Center personnel who participated were assigned roles corresponding to the acquisition jobs called for in DOD-STD-2167. One person was assigned the job of "customer," responsible for seeing that MIMSY can perform the functions required. As an aid to the design process the customer was assumed to be knowledgeable of Ada and formal methods as well as MIMSY's functionality. While this may not always be a realistic assumption in real acquisitions, having a knowledgeable customer limited the amount of training that would have been needed to convince the customer of the need for Ada and formal methods. The customer had an "advisor" to provide technical guidance; this advisor corresponded to the kind of support that MITRE would supply to a real customer.

The remaining participants in the MIMSY exercise were formed into a "contractor" team corresponding to an independent company that had been awarded the MIMSY contract. Five people, all with some experience in using Ada and formal methods, made up the team that actually designed MIMSY. One person had the job of "project leader," with overall responsibility for getting the design and the documentation done on time. Another person had the responsibility for Configuration Management (CM) and Quality Assurance (QA), corresponding to an independent CM/QA department within the contractor. Three other people were the designers of MIMSY. They worked collectively designing the earliest stages of MIMSY, and then later worked separately on designing a major component of MIMSY.

Most of the documents called for in DOD-STD-2167 for the early stages of a project's development were produced. This included a Statement Of Work (SOW), a Software Development Plan (SDP), a Software Requirements Specification (SRS) and a Software Top Level Design Document (STLDD). While the SDP and the SRS were primarily English text, the STLDD was primarily Ada PDL text. Both iterations of the STLDD were approximately 40 pages long, with 85% of each document being Ada PDL.

To more easily use formal methods in the design process, a PDL was chosen that supports a wide variety of formal methods. The ANNA [Luck84] language was chosen because it is based on Ada and because ANNA has support for formal methods missing in Ada through structured comments. Work is currently underway at Stanford University on tools that can recognize ANNA's structured comments and produce code to check whether design constraints are being met. While the

MIMSY team did not have access to these tools, ANNA was used anyway because it provided a precise notation for expressing the design. The current version of ANNA has no support for Ada tasks but a tasking constraint language called TSL [Helm85], written by the same computer scientists who built ANNA, has been written that does support design using Ada tasks. MIMSY used TSL together with ANNA whenever Ada tasks were used in the design.

The design methodology that was used on MIMSY was Buhr's system design methodology [Buhr84] with Ada. While this design methodology makes heavy use of the Ada language and the formal methods which Ada directly supports, Buhr's methodology does not support the formal methods that ANNA adds to Ada. Rather than trying to develop a design methodology specifically for ANNA or using a design methodology specifically created to use a wide range of formal methods, MIMSY tried to use ANNA's formal methods with an existing design methodology. One goal of the MIMSY project was to see how well formal methods could be used with existing design methodologies.

Formal reviews of the MIMSY documentation were conducted in accordance with DOD-STD-2167. The SDP and the SRS were reviewed at a Software Specification Review (SSR). Two different Preliminary Design Reviews (PDRs) were conducted on the STLDD. The second PDR was needed to add detail and answer customer questions that were raised at the first PDR. In addition to the MIMSY participants, outside MITRE personnel were invited to these reviews so more people could see the advantages/problems of using Ada, ANNA, formal methods and DOD-STD-2167 together.

## SECTION 2

### MIMSY'S FUNCTIONALITY

#### 2.1 OPERATIONS TO BE PERFORMED

When completed, the MIMSY system will allow a user to control the output of one or more Ada applications programs on the screen of a terminal. MIMSY will provide a primitive horizontal windowing capability that allows a user to set aside certain lines of the screen for the output from an Ada program, and then be able to scroll through the output of a particular window. The user interface and appearance on the screen are similar to that of DEC's "lsedit" editor [DEC85a] or the "emacs" [Stal85] editor.

In addition to a primitive window manager, MIMSY allows the user to control the Ada applications programs that are creating the output being displayed in the windows. A user can create, suspend, resume and terminate an Ada applications program under MIMSY's control. While MIMSY was required to be portable to a number of different Ada operating environments, both MIMSY and the applications programs that it is controlling were assumed to be implemented first on top of DEC's VMS operating system. Development of MIMSY would be done in Ada using the DEC Ada system [DEC85b].

MIMSY's functionality is a subset of the functionality provided [TI83]. AIM allowed control of the input to the applications program as well as the output from them. AIM also provided a user interface to the full CAIS [KIT84] functionality while MIMSY provides only a small subset. With the completion of the AIM project, it will be interesting to compare its approach with the more formal approach taken with MIMSY.

#### 2.2 MAJOR MIMSY COMPONENTS

There are four major components to the MIMSY system. The interfaces to and operations performed by these components are discussed in detail in MIMSY's STLDD. A brief summary of these components' functionality is included below as a point of reference for this presentation.

The Keyboard Controller processes all the keystrokes typed at a user's keyboard. This component handles functions such as echoing, command line editing (backspace key) and determination of command

completion (newline character). Completed commands are sent to the next component for analysis.

The Command Interpreter validates any user commands for correctness. MIMSY has a few well-defined commands. Any misspellings or non-existent commands are reported as errors to the user through a message displayed on the terminal's screen. The user often refers to previously created screen windows and applications processes in commands; the Command Interpreter ensures that these names are still valid references.

The Structure Manager encapsulates the control of the main data structures in MIMSY. This includes the data structures controlling the positioning and contents of the windows on the screen, and the structures that control the applications programs. MIMSY makes extensive use of tasking in these data structures to maximize concurrency between the applications programs.

The CAISette component is an attempt to provide an operating system-independent interface to the underlying support environment. The CAIS [KIT84] is an example of such an interface for Ada programs, but MIMSY did not need all the CAIS functionality. To minimize the amount of work that would be needed on this interface, a CAISette was defined that contained only those CAIS functions necessary to support MIMSY. The CAISette defines not only the functional interface but also the behavioral model of the underlying support environment, so anyone porting MIMSY to another environment will be aware of what MIMSY assumes about its environment.

## SECTION 3

### DESIGN METHODOLOGY USED WITH MIMSY

#### 3.1 DESCRIPTION OF DESIGN METHODOLOGY

The design methodology used during the design phase of MIMSY was an actor and object oriented methodology similar to those used in SMALLTALK [Gold83] and PAMELA [Cher85]. The design methodology makes extensive use of Ada tasking, with active tasks initiating operations and passive tasks providing services. Each of the major MIMSY components was designed as a task, with the task entry points and parameters defining the messages that could be passed between tasks. Major objects within each component were also designed as tasks, although the details of the tasking structure within a major component were hidden from the other components.

An important part of the design methodology is to identify which tasks are active and which are passive. Since MIMSY's primary function is to respond to commands typed by a user, the KEYBOARD CONTROLLER component was made the primary active object. All the other components of MIMSY serve the requests generated by the KEYBOARD CONTROLLER (and therefore the user).

MIMSY is required to manage the user's terminal screen, providing multi-windowing for multiple user programs. This requires the management of data flows to and from multiple concurrent user programs, yet at the same time managing the display of a terminal screen that must be updated sequentially. To minimize the complexity of the data flows in MIMSY, the design methodology calls for each of the major data structures to have its own task to manage it. The STRUCTURE MANAGER component of MIMSY is responsible for the data structures that control the windows on the screen and the user's programs. This design methodology leads to a STRUCTURE MANAGER that is a hierarchy of data structures and objects (and their tasks) that are dynamically created and destroyed (terminated) in response to user commands.

The design methodology uses extensive data abstraction, where all the data structures and objects are defined in terms of the operations that can be performed on them. Data abstraction calls for restrictions to be placed on the operations on the data types. In MIMSY's case these restrictions had both a static and a dynamic component. Static restrictions are the classical enumerations of the operations that are allowed and the allowable range of values on the data types. Dynamic restrictions depend on the current state of

the data types and objects. These dynamic restrictions are typically represented as axioms that relate previous states with what are now legal operations on the data types. In a system like MIMSY, the extensive use of tasking can lead to rapidly changing states. The design methodology tries to limit the complexity of the data type restrictions by structuring the objects (and their tasks) so the restrictions can be expressed in a hierarchy.

### 3.2 DESCRIPTION OF DESIGN NOTATION

The design methodology described in the previous section results in a variety of design products that are used to capture design decisions for the benefit of the coders, reviewers (customer), designers and future maintainers. Two different design notations were used to capture MIMSY's design. These two design notations try to find a balance between a high level pictorial representation of the design (used to rapidly convey the essence of the design) and a detailed description of the design (used to document detailed design decisions).

The highest levels of MIMSY's design was represented with Buhr diagrams [Buhr84]. Buhr diagrams provide a series of graphical icons that may be connected to show the flow of control and data in a design. The complexity of a design (i.e., cluttered pictures) can be limited through hierarchies of Buhr diagrams. Buhr defines a one-to-one mapping between the icons of the graphical representation and the Ada code skeletons that can represent the icons. The icons for Ada tasks are especially expressive, allowing a designer to capture the behavior of tasks by graphically expressing the nature of a task's entry calls (selective entry, timed wait, etc.).

The lower (and more detailed) levels of MIMSY's design were represented in ANNA text [Luck84]. While Buhr diagrams can be used to express the structure of the Ada packages, tasks and subprograms, the diagrams do not allow the structure of data or the restrictions on subprogram calls to be expressed. Preconditions, postconditions, exception propagation rules and other detailed design decisions were captured in ANNA text.

Currently the ANNA language cannot be used to express restrictions on the actions of Ada tasks that might exist in the design. This shortcoming has been overcome by ANNA's companion language, the Task Sequencing Language (TSL) [Helm85]. TSL was defined as an extension to ANNA, so TSL and ANNA text may be combined in the same design. TSL allows restrictions on the actions of Ada tasks to be defined, much like ANNA allows restrictions on the actions of

subprograms and packages to be defined. MIMSY's designers used ANNA and TSL together to capture the details of how the system was to function.

ANNA has no support for capturing the design management information called for in DOD-STD-2167, so MIMSY's designers invented their own constructs for capturing such information. In MIMSY's case, structured comments were used with the '/' character used as the sentinel character. Design management information, such as requirements traceability and change logs, consisted of a reserved keyword (such as "Change Log") followed by a natural language description of the design management information. Additional support tools could have been built to extract this information and create management reports and/or the documentation called for in DOD-STD-2167.

MIMSY's designers worked from a high level description of the system to a detailed description. MIMSY's design was first documented at a high level with Buhr diagrams. A single high level diagram of the MIMSY system was broken down into a Buhr diagram for each of the four major MIMSY components. Each of the component's Buhr diagrams was further broken down into more detailed Buhr diagrams. The most detailed Buhr diagrams describe all the tasks that exist in MIMSY and all the major subprograms that implement MIMSY functionality.

With the graphical representation of the design complete, the designers then began to transcribe the diagrams into PDL text. The Buhr diagrams have a mapping to Ada constructs defined for them, so the designers had little trouble creating Ada package and subprogram specifications from the diagrams. The designers then added the ANNA and TSL text into the Ada text created from the diagrams. The ANNA and TSL text captured the increased detail, restrictions and system behavior that could not be represented in the diagrams.

The ANNA and TSL text had the same organization as the MIMSY components and subcomponents, so each component had its own collection of ANNA and TSL text that described its design. The most common method used by a MIMSY designer to show that the ANNA and TSL text for a component was consistent was proof by induction. Most of these inductive proofs used a few basis cases then recursive induction steps that were similar to Lisp code.

### 3.3 RATIONALE FOR USING THIS DESIGN METHODOLOGY

The design methodology and design notations used in MIMSY provided a good match between the needs to design a system and fulfill

the original requirements of the MIMSY investigation (as outlined in Section I of this document). The methods and notations were very "Ada-like" in that the syntax and semantics of the Ada language was used as much as possible. The addition of the formal notations of ANNA and TSL just completes the formalization process begun by using Ada as a PDL.

The design methodology used in MIMSY was chosen because it allows the system behavior as well as the system's structure to be expressed in Ada PDL. Used properly, Ada tasking can be the most expressive notation to describe a system's behavior. MIMSY has extensive requirements for the independence and concurrency of the windows, user tasks, keyboard, etc. The Ada language defines almost every construct of the language except task entries as being re-entrant. One goal of the designers was to make the components of MIMSY as reentrant as possible, with those parts of the system that must have their actions serialized controlled by tasks. The design methodology's use of the Ada task as the primary design paradigm does not artificially constrain the designers in their use of Ada.

Both the Buhr graphical notation and ANNA/TSL make extensive use of the syntax and semantics of the Ada language. Their constructs are the same as the constructs provided in the Ada language. Neither is an attempt to graft some other design paradigms or constructs on the Ada language; as a result, designers who are familiar with Ada have less difficulty in working with Buhr and ANNA. If Ada is really going to be used as a PDL, then any annotations and graphical representations used with Ada should be based on the Ada language.

DOD-STD-2167 implies the use of a top-down design methodology, although it is possible to use it with other design methodologies such as bottom-up. The method of starting with increasingly detailed Buhr diagrams and then moving to Ada/ANNA/TSL text fits the description of a top-down methodology. Since many systems are also developed using a top-down methodology, the lessons learned in MIMSY have some applicability to real system designs. This may change in the future when large libraries of reusable Ada (or ANNA) packages allow many systems to be designed from the bottom up.

Many existing design methodologies and notations make extensive use of graphics. Designers find graphics to be a convenient way to express a system's design in just a few pages, particularly to non-technical reviewers. The introduction of Ada as a PDL will not (and should not) change a designer's preference for the use of graphical notations, especially at the highest levels of a design. MIMSY chose to use graphical notation in its designs. Buhr diagrams were

chosen because they were the most expressive notation that was well-integrated with the Ada language.

The use of ANNA and TSL to formally define the restrictions and behavior of MIMSY provided the designers with an opportunity to see how the increased use of formal methods might improve a design. One problem with many designs is that important design decisions are either ambiguously captured or are not written down at all (remaining inside the designer's head). When a coder or maintainer attempts to change the system at some point in the future, the lack of complete design documentation may lead to the introduction of errors and/or excessive maintenance costs. One of MIMSY's major requirements was that it be maintainable; the introduction of formal methods into the design is an attempt to make the system more maintainable. ANNA and TSL were created to improve Ada's support for formal methods. MIMSY's designers decided to use them as they provided the best opportunity to increase formalism in Ada PDLs.

## SECTION 4

### RELATIONSHIP OF ANNA TO FORMAL METHODS

#### 4.1 ANNA IMPROVES ADA'S COVERAGE OF FORMAL METHODS

While Ada's constructs provide support for many formal methods, there are some formal methods that Ada does not cover. ANNA tries to improve Ada's coverage of formal methods by providing new and modified constructs that a designer or coder can use with Ada. The constructs described below allow a designer to use some of the classical techniques of formal methods with systems written in Ada.

ANNA provides the constructs to allow the use of first order predicate logic in Ada. The "exists" and "for all" quantifiers allow a designer to formally specify restrictions on an Ada data type, important if abstract data types are to be used extensively. ANNA allows the use of logical implication and equivalence operators in quantified expressions.

While Ada's strong type checking provides one level of preconditions and postconditions, ANNA provides the constructs for the full use of preconditions and postconditions beyond type checking. Annotations can be placed on individual statements (as in classical weakest precondition notations), on groups of statements (e.g., for loop invariants) and on Ada subprograms and packages. ANNA provides constructs for defining the propagation of Ada exceptions, which allows both the designer and user of an Ada construct to know exactly what conditions may lead to an exception and what the state of the software will be if exceptions are propagated.

Ada packages will tend to be the primary means of organizing an Ada program. Systems built with Ada will be hierarchies of (re-usable) Ada packages. Just as packages are the primary building block of Ada, packages are also the primary building block of ANNA. The main ANNA constructs for describing restrictions and system behavior are package annotations, so there is no separation between the primary building blocks of Ada and ANNA.

ANNA provides the additional constructs to allow axiomatic descriptions of a package's behavior to be built. These axioms can be used to specify how the subprograms, data types and objects of a package relate to one another. Mathematical induction can be used to define the actions of a package after many state changes have occurred.

## 4.2 ANNA IS WELL-INTEGRATED INTO ADA

ANNA was designed, constructed and documented to allow ANNA annotations to be fully integrated with MIL-STD-1815A Ada [DOD83]. ANNA's reference manual [Luck84] has the same organization [DOD83]. All the new ANNA constructs are defined as extensions to Ada's syntax and semantics. Note that ANNA inherits some of Ada's shortcomings as well. For example, Ada programs have been shown to be very difficult to formally verify for correctness because of Ada's support for type attributes, tasking, etc. ANNA was not designed to solve all of Ada's problems; instead ANNA provides more annotations to the existing Ada semantics to provide improved formal methods support.

All ANNA constructs are defined as a syntax that appears inside Ada comments. Unique sentinel characters are placed after the Ada "--" comment indication. For ANNA this sentinel character is '|' (or '|') and ':', for TSL the character is '+'. This means that an Ada compiler is unaffected by ANNA's and TSL's statements because they are hidden inside comments. Only ANNA/TSL tools (and human readers) use these statements to further explain the program.

ANNA provides constructs for annotating almost all of Ada's constructs, tasks being the major exception. Ada's access types, records, exceptions, generics, packages and other constructs all have ANNA constructs associated with them. Because ANNA (with TSL) covers the entire Ada language, designers and coders using Ada are not tempted to use subsets of Ada in an attempt to stay within the annotation language's limitations. ANNA does not try to "glue on" the formalism of some existing formal methods system; ANNA has been created just to deal with Ada. As a result, designers who understand Ada's model of computation will have less difficulty using ANNA than trying to use some other annotation language that was originally built for a different model of computation.

The semantics for the elaboration, visibility, scope and other attributes of ANNA statements are the same as for the Ada constructs they are annotating (TSL uses different scope and visibility rules). The ANNA reference manual [Luck84] shows that each ANNA statement could be replaced in-line by Ada statements that would actually do the checking done by the ANNA statement. This mapping would not be possible if ANNA statements were outside Ada's semantics.

In many instances the restriction a designer might want to place on the behavior of an Ada construct might be expressed through some additional Ada text. ANNA allows designers to place additional Ada text (called "virtual Ada") in Ada designs. Virtual Ada has the

same syntax and semantics as regular Ada, the only difference being that virtual Ada is placed in structured comments that have the ':' character as the sentinel character. Virtual Ada can be used in combination with ANNA text; together they can be used to create very expressive specification of the non-commentary Ada text.

#### 4.3 TSL COVERS ADA TASKING

Currently the ANNA language does not cover Ada tasks. Instead, tasks are covered by a separate but compatible annotation language called TSL. TSL could be used only with Ada tasking constructs (task, task types, entries) but recent changes to TSL allow it to be used with other Ada constructs such as subprograms.

TSL assumes that all the "events" associated with Ada tasking can be placed into a single stream. This stream of "events" would contain all the task creations and terminations, entry calls, accepts and rendezvous, etc. TSL allows the designer to restrict the ordering of events in the event stream. These restrictions can be used by a designer to prevent common concurrency problems such as deadlock. TSL annotations can also be used to better specify the conditions that a task expects to see at some point in its execution.

While ANNA is oriented to both static and dynamic analysis of Ada programs, TSL is oriented almost entirely towards dynamic analysis. The non-determinism in Ada's tasking model makes it very difficult to predict ahead of time the exact ordering of events in the Ada task event sequence. TSL provides a notation that allows run-time monitors to catch incorrect sequences of task events.

Because there is only one task event sequence, the sequence may contain events of tasks that are not of interest to TSL statements. TSL handles this and other complications of checking the sequence of events in much the same way as Prolog [Cloc81] handles the search of its data base. As with Prolog, TSL statements define a "pattern" that is matched against the actual contents of the event sequence. TSL has operators similar to Prolog's "cut" that can be used to limit the complexity of the event sequence matching. This contrast between TSL's Prolog-like approach and ANNA's predicate calculus approach helps explain why ANNA and TSL are currently two separate languages.

While the ANNA language's definition [Luck84] is stable at this point, the language definition of TSL [Helm85] is still subject to change. For example, based on the experiences of early TSL users (such as MIMSY), TSL has added user-defined "events" to the type of

events that can be placed in the task event sequence. This allows a user to place events such as certain subprogram calls or changes to key objects with the other tasking events such as accepts and releases.

#### 4.4 ANNA USABLE WITH DIFFERENT APPROACHES

There are many different approaches to using a formal method to design software. For example, the use of data abstraction can be broken down into the algebraic specification approach versus the abstract model approach [Berg82]. Each approach has its own advantages and disadvantages; ANNA can be used with either approach.

The algebraic specification approach builds the description of the data abstraction from low-level axioms. ANNA package axioms can be used to provide these low-level axioms for an Ada abstract data type. Since ANNA's package axioms use a form similar to classical data type axioms, a designer experienced in the classical forms has little difficulty in learning to use ANNA's axioms.

The abstract model approach builds the abstract data type from previously defined (and proven) data types and concepts. These previously defined data types provide a model on which the new abstract data type is built. An Ada programmer might view this approach as an example of reusability.

ANNA supports the abstract model approach by reusable Ada libraries that encapsulate (or model) the concept needed by the abstract data type under construction. For example, a library can contain the Ada functions and ANNA axioms to check whether an abstract data type has the properties of a group or monoid. This library would be built as a "virtual" Ada package, allowing for reusability. If designers have some of these reusable libraries at their disposal, the specification of the properties of a new abstract data type becomes less verbose than if all the individual axioms had to be repeated.

In formal methods the full specification of how a system is supposed to work is built up on layers of theorems and lemmas. The proof that the system meets its specification can be done by first proving the lower level theorems (by showing that all the axioms and conditions hold) and then showing that the lower level theorems prove the correctness of the higher level theorems and then finally the entire system. ANNA supports this approach because the software can be built with layers of Ada packages. If the lower level packages can be shown to be consistent with the higher level ones, the entire system can be shown to have the correct behavior.

#### 4.5 AN ANNA EXAMPLE FROM MIMSY

Appendix A of this document (Figures 1 and 2) contains an example of some ANNA text that was written for the MIMSY project. In this case, it is part of the annotation for the STRUCTURE MANAGER component of MIMSY. The text shown is part of the package specification for STRUCTURE\_MANAGER; the real package specification is more verbose. Some of the details have been eliminated so the highlights of the package can be seen.

In this example the preconditions and axioms are trying to show that the first thing that a user of this package must do is call START; attempts to ANALYZE a COMMAND or TERMINATE are illegal. One of the things the START procedure does is establish the MAXIMUM NUMBER OF screen VIEWPORTS and user PROCESSES that can exist at any one time. Once these numbers have been established, they must not be changed by a call to ANALYZE COMMAND. Calls to START after the initial call are illegal. A call to TERMINATE ALL prevents any further calls to the main procedure of STRUCTURE\_MANAGER, which is ANALYZE\_COMMAND.

Note the use of induction inside the axioms for this package. The two axioms for ANALYZE\_COMMAND show that the maximum number of viewports and processes remain the same no matter what kind of ANALYZE\_COMMAND is issued. The "for all C : COMMAND\_RECORDS" allows this to be stated for any type of command that was analyzed.

#### 4.6 ANNA TOOLS

Currently ANNA is useful primarily as a rigorous notation that designers can use to specify the behavior of Ada software. ANNA's situation is similar to that of Ada's before the availability of robust, validated Ada compilers. The rigor of the notation makes it useful to those software engineers who are conversant in the language, but wider use must await the necessary software tools.

In ANNA's case these tools are currently under development at Stanford University and in Europe under the ESPRIT program [ESPR85]. These tools will allow the ANNA text, currently hidden from the Ada compiler, to be analyzed. An early version of these tools would check the syntax of the ANNA for syntactic correctness. This early tool would create a DIANA [Goos83] syntax tree (in addition to checking for simple errors). The DIANA tree would be used as the input into later, more detailed analysis tools. Such an analysis tool would convert all the ANNA statements (as well as all the virtual Ada statements) into compilable Ada statements. The entire design is then compiled, linked and executed just like any other Ada

program. The execution of the design (using Ada's execution model) provides additional checks on the validity of the design. If an ANNA constraint is violated, the predefined exception "ANNA EXCEPTION" is raised in the program unit where the constraint is violated.

To provide designers and coders guidance with the use of ANNA and its tools, several large projects have begun that will build programming environments that support the software lifecycle, including the use of ANNA as a PDL. The PROSPECTRA [ESPR85] project in Europe is the largest example of this. These environments would integrate ANNA tools with other lifecycle tools. The development of these tools is still some time in the future, although the MIMSY project showed that ANNA and TSL as a notation is useful to a project when few or no tools are available.

#### 4.7 TSL TOOLS

Also under development are the TSL tools that allow for run-time checking of the Ada task event sequence. These tools involve changes to the Ada support environment to create and manage the task event sequence [Helm84]. These tools also make sure that Ada tasks place their events into this sequence.

The introduction of TSL statements will provide the (modified) Ada support environment with constraints to check against an actual tasking sequence of events from an executing design. As with ANNA, the primary method of checking the design becomes the execution of the design itself. Longer range goals would add additional design checking mechanisms such as symbolic execution to ANNA and TSL. In either case, the use of executable designs will be a sharp contrast from the state of the practice in the use of Program Design Languages.

## SECTION 5

### MIMSY'S USE OF FORMAL METHODS

#### 5.1 BUHR DIAGRAM'S RELATIONSHIP TO FORMAL METHODS

As discussed in earlier sections, MIMSY used a design methodology that used Ada tasks as the primary design paradigm, with Ada's tasking model used as the mechanism for data and control flow among the entities of the design. MIMSY's designers tried to follow good design practices by using hierarchies to hide details of the design until lower levels when the details are necessary. In MIMSY's case Buhr diagrams were constructed with incremental refinement, where a top level diagram was refined into lower level Buhr diagrams that contained more and more detail.

One complication that the designers encountered was how to refine one Buhr diagram into its lower level diagrams. On one hand, the designers wanted to use traditional information hiding, placing only the minimum amount of detail necessary in a higher level. On the other hand, enough detail had to be provided to higher levels of the design to provide enough information to satisfy the formal methods being used. A system built with hierarchies will want to prove that important attributes and behaviors are maintained among levels (and so the entire system). The information needs of formal methods and data abstraction/information hiding can come into conflict.

One example of where this conflict arose was in the Structure Manager component in MIMSY. Using the design methodology lead to a decision to use dynamically created and terminated tasks in the structure manager to control the windows and processes a user might create. Initially the designers hoped that the implementation details of using tasks within the body of Structure Manager could be hidden from the caller. Yet the ANNA and TSL statements that would be appearing in the package specification for Structure Manager needed to know something about the behavior and restrictions being implemented in the body. Eventually the users and designers of Structure Manager had to compromise about how much information is visible in the package specification.

When the designers using Buhr diagrams go from one level of detail to the next, they need to specify the behavior of the lower levels as well as the interfaces. The icons in the Buhr diagrams have good support for specifying the interfaces between levels. Support for behavior is not as strong; some tasking behavior can be

described in the diagrams but not much else. As designers moved from the Buhr diagrams to the more detailed presentation provided by ANNA and TSL, they wanted to capture the intended behavior to be implemented in the lower levels of the design without unduly constraining the eventual designers and coders of the lower levels.

An example of this occurred in MIMSY with the CAISette component, which encapsulated the interface to the underlying Ada support environment. Since many different components would be using CAISette to access the support environment, there were many assumptions that designers made of CAISette that the eventual coders of CAISette would have to consider. Some of these issues are discussed in detail later. A compromise used in MIMSY was to use virtual ANNA and TSL statements to define the intended behavior of CAISette. The eventual coders of CAISette are not bound to implement the virtual Ada constructs, as long as the behavior of the CAISette implementation matches the behavior of the virtual ANNA.

The hierarchy of Buhr diagrams containing MIMSY's design had different designers creating the different levels. This required that the designers be very precise in their designs for both the diagrams and the eventual translation to ANNA/TSL text. The need for precise specifications of the components would have been true if MIMSY had been designed from the bottom up with reusable Ada packages. A package could not have been "reusable" unless its specification provided enough behavioral and interface information to allow a designer to determine how (and if) this package could be reused in the current design. Designers should be able to assess the reusability of a package without having to read all the details of that package's body.

## 5.2 MIMSY USED ALGEBRAIC SPECIFICATION APPROACH

As discussed earlier, MIMSY could have chosen to use either the algebraic specification or the abstract modeling approach to abstract data types. The abstract modeling approach could have been useful to MIMSY in designing several of the components. For example, the CAISette package controls the display of lines of text on the user's screen. The control and display characters necessary to properly display a line of text on the screen in its proper position could have been modeled on sequences.

If a pre-existing data type called "sequence" existed, CAISette could have based the messages (characters) that are sent to the terminal on the sequence data type. The sequence data type could be constructed such that its axioms proved that the sequence did not allow messages (characters) to become interspersed. Then the only

thing that the designer of CAISette would have to do would be to show that CAISette used the sequence data type correctly in the new data types which controlled the user's screen.

Unfortunately MIMSY did not have any such "sequence" data type to use in modeling the new data types needed for CAISette and the other components. Without a library of reusable abstract data type models (or the time necessary to create them), MIMSY used the algebraic specification approach. As a result, components such as CAISette convey the same amount of information about their restrictions and behavior, but these specifications are much more verbose than if simple references to reusable data modeling packages had been used.

Mature tools and notations used with data abstraction, such as AFFIRM [Gerh80], come with a library of reusable lower level data types. AFFIRM has an extensive set of data types and theorems for dealing with sequences. The library of sequence data types makes it easier for an AFFIRM user to create (model) their own data types, and the library of sequence theorems provides an AFFIRM user with a ready list of important properties that the user's data types should satisfy.

ANNA users need to develop their own libraries of reusable abstract models. The models that might go into this library (sequences, numbering systems, etc.) will depend on the applications that the users are designing systems for. Users must weigh the costs of creating and maintaining such a library versus the future benefits in savings to later designs.

### 5.3 MIMSY USED ANNA AND TSL AS THE PDL

MIMSY's designers used a combination of ANNA and TSL as the Program Design Language (PDL) for MIMSY. MIMSY's design was done to the Preliminary Design level of detail. As a result, the Preliminary Design document consisted of the Buhr diagrams and Ada package specifications (with the ANNA and TSL statements placed in the package specification). Only a limited amount of information was placed in the package bodies, so information on how the bodies of the packages will be designed is deferred until Detailed Design.

MIMSY's designers did not have access to any ANNA or TSL tools, not even to tools that would have checked the statements for syntactic correctness. An Ada compiler was used to check Ada statements, but the ANNA and TSL statements were hidden from it in comments. The ANNA/TSL statements were used as an expressive notation for formally specifying the design's restrictions, assumptions

and behavior. Any checking or "execution" of ANNA or TSL statements had to be done by hand.

One tool that was available to the MIMSY designers was a modified version of Digital Equipment Corporation's language-sensitive editor "lsedit" [DEC85a]. This structured editor can use a user-defined template that defines the syntax of a programming language. The editor comes with a template (that is similar to a Backus-Naur Form syntax definition) for the major languages supported by the computer. MIMSY's designers modified the template for Ada to include additional syntax definitions for ANNA's structured comments. Once the editor's basic functions were mastered, the additional support for ANNA did not require learning any additional commands. While the use of this editor was no guarantee of semantic correctness of the code, its template-driven style did allow MIMSY's designers to concentrate on the details of the design instead of on the details of ANNA's syntax. Users of an expressive, Ada-based PDL such as ANNA should consider using a good editor in the creation of their text to limit the number of simple syntax errors that might otherwise creep in.

Because the design team did not have any experience in using ANNA, a series of informal classes and seminars were held before the design effort began. Fortunately, MIMSY's designers had a good educational background in the formal methods being used in MIMSY. Everyone was familiar with preconditions, postconditions, axioms, induction, recursion, etc. The training consisted of reading and then discussing the ANNA and TSL reference manuals as well as some of the classic papers in data type abstraction by Guttag [Gutt78]. This training reduced the amount of confusion and misconceptions that would have otherwise damaged MIMSY's design. The use of formal methods and notations in a design should be preceded by a careful analysis of the current levels of experience in the formal methods to be used and the formal notations and tools that will be part of the design effort.

One aspect to the quality of a design is the degree to which the design considers all the "special cases" that might occur. Examples of special cases include initialization, shut-down, overloading, illegal inputs, etc. The ANNA and TSL languages have the rigor and notation for expressing all these special cases. Once a designer has mastered all the ANNA and TSL constructs that could be applied to an Ada construct (such as exceptions, packages, tasks, etc.), the designer tries to be as expressive as possible by seeing if an Ada construct needs to have all its associated ANNA or TSL constructs included with it. This naturally leads the designer to consider special cases such as initialization and termination.

This tendency may have been helped by using the structured "lsedit" editor. In using the editor the designer is presented with a menu of choices for which (ANNA) construct is to be used. Currently this menu corresponds to the ANNA BNF syntax that defines which kinds of ANNA statements can be associated with which kinds of Ada statements. Often the different ANNA statements correspond to how special cases are to be handled. A more powerful version of such an editor might organize its menu explicitly by special cases so the designers are further encouraged to consider these cases.

During a formal review of a design, the reviewers will demand to see evidence that important issues such as steady state operations, system response to commands and special cases have been addressed. The use of ANNA and TSL and the PDL provided proof to the reviewers that these issues were considered. Because the use of formal methods and a rigorous notation such as ANNA should limit the amount of ambiguity in the design, the reviewers, designers and managers (as well as future maintainers) should all be able to read MIMSY's PDL and come away with the same understanding of the system.

#### 5.4 USE OF ANNA WITH CAISETTE PACKAGE

The interface to the underlying support environment (such as the operating system) was encapsulated in the CAISette package. MIMSY's CAISette package is based on the CAIS [KIT84] interface standard. Only those CAIS functions needed by MIMSY were included in CAISette; other CAIS functions were not included.

One problem with the CAIS specification is that it does not fully define the restrictions and behavior of the subprogram that might be called. If MIMSY's CAISette package were defined to the same level of detail as the CAIS specification, it might look something like the first package (Figures 3 and 4) in Appendix B. All the examples in Appendix B are for a small part of the CAISette package, in this case from the scrolling terminal. Only the highlights of this package are presented.

Figures 5 through 8 in Appendix B replace the informal comments from the first example (Figures 3 and 4) with ANNA statements that define when exceptions are raised, etc. This second example is less ambiguous than the first. The second example provides a much more detailed interface for CAISette's users and designers. What is still missing are the details of the behavior of a scrolling terminal. The second example does not answer important questions such as whether concurrent output to the terminal is serialized or is interspersed.

Figures 9 and 10 in Appendix B contain just TSL statements for the scrolling terminal package. To save space, the ANNA statements from the second example are not repeated. In the actual CAISette package specification both ANNA and TSL statements are used.

This third example uses "virtual Ada" to define a task type that controls the output requests to an open terminal. The messages being sent to the screen (consisting of cursor positioning commands and the characters to be displayed) are serialized and output one at a time. This means that the user of CAISette does not have to worry about two tasks sending a string to the screen and having the strings being interspersed. One string will be output in its proper position, then the other string will be output in its position.

Note that the use of the "virtual" task does not force the body of CAISette to be implemented with a task to serialize screen output. For example, some Ada support environments will, by default, serialize the output requests so the programmer does not have to add any additional code to achieve this behavior. If MIMSY is ever implemented on a support environment that does not serialize its output, then the "virtual" tasks would likely be implemented with real tasks in the body of the CAISette package. This is another example of information hiding and design decision deferral until necessary.

## 5.5 ADVANTAGES OF USING FORMAL METHODS IN MIMSY

The use of the formal methods described in this document on MIMSY lead to benefits for both the current designers and reviewers of MIMSY, as well as for future coders, users and maintainers. As with many of the proposals for improving software engineering practices, the increased use of formal methods should increase the quality and decrease the overall life cycle costs of software. The use of formal methods in this project should help MIMSY in achieving these goals.

By using a formal notation as the PDL, MIMSY allows for the possibility of design analysis by ANNA and TSL tools when those tools become available. One of MIMSY's requirements was that it should support portability to other support environments and allow more functions to be added. These future upgrades should be less costly and time-consuming if the maintainers can analyze the effects of their changes (possible introduction of errors to existing code, new and excessive performance delays, etc.) before the changes are committed to code. Future ANNA tools should allow for the creation of executable designs. If the maintainers can begin the maintenance cycle by working on the design documents (instead of working from

the implementation code), maintenance is more likely to result in a usable system at less cost.

One technique used to validate system characteristics such as the user interface is the construction of a rapid prototype. This rapid prototype might be used by the customer during a formal review to determine if the user interface is satisfactory. If the customer finds (through hands-on experience) that the user interface is satisfactory, then the customer will approve the interface, and expect to see a similar interface in the final system.

One potential problem with rapid prototypes is that unless project management control is exercised, the rapid prototype and the actual system can become two divergent products. While developed independently, the rapid prototype and the actual system must remain compatible (have the same user interface behavior for example) or else the knowledge (and customer acceptance) gained from the rapid prototype cannot be inserted back into the actual system. Since the customer will have used and approved the rapid prototype, the customer may end up preferring the rapid prototype over the actual system.

Part of MIMSY's development plan calls for the development of rapid prototypes to validate the user interface for controlling the windows and user processes. By rigorously defining the interfaces and behavior (with ANNA and TSL) of MIMSY, rapid prototype builders and the designers/coders should be able to proceed from the same high-level design document and not end with incompatible products. The system behavior of the final MIMSY product should be the same as the behavior of the rapid prototype shown to the customer.

One way of validating a product is to build test cases that provide inputs to the system as well as to define what the expected outputs should be. Typically these test cases are constructed from the original system requirements. These test cases are applied to the completed system during final customer acceptance tests. If the tests are successful then the system is accepted.

The use of a formal specification and design language (such as ANNA) should provide a more precise definition of system behavior. This should lead to the construction of more comprehensive test cases. Since the use of ANNA and TSL (used with a good ANNA tool set) can produce executable designs, this allows the application of test cases at the Preliminary and Detailed Design Reviews as well as the final customer acceptance review. The early applications of these test cases can identify problems with the design early in its life cycle, when such errors are cheaper to fix than during final testing.

The use of a formal design notation such as ANNA should reduce the possibility of different interpretations of the design documents. Already this section has shown how the same design document might be used by the designers, coders, rapid prototype builders, test case writers and customers. If the design documents were in a natural language, each user of the document might read something different from the ambiguous text. A formal design notation can reduce the possibility of this occurring.

## 5.6 REVIEW PROCESS WHEN USING FORMAL METHODS

The preliminary design documents, consisting mostly of ANNA/TSL text and supporting Buhr diagrams, were reviewed by the "customer" at a Preliminary Design Review. As with most formal reviews, the design documents were distributed to the attendees ahead of time for their review. The actual review consisted of a presentation of the design by the design team, followed by questions and comments on the design by the customer.

The primary medium for both the presentation of the design and the customer questions about the design were the ANNA axioms and pre/postconditions. The discussion on how MIMSY would operate in steady state and exceptional conditions was done by tracing the flow of control and data through the axioms. Customer questions about the current state of the system or the assumptions being made by a MIMSY component were answered by referencing the appropriate axiom. Special cases are discussed by tracing the raising and propagation of exceptions. The use of ANNA as the design notation eliminated many of the questions about what a particular sentence of the design document really means.

The design review of MIMSY went as well as it did because the customer reviewers had a good background in both the formal methods being used and in the ANNA and TSL languages. As a result, the customer did not require a lot of education before the design documents were delivered. While the use of a formal notation such as ANNA on MIMSY requires high skill levels of both the designers and the customer, the advantages gained from an unambiguous design offset any costs from using skilled people on the project.

## SECTION 6

### CONCLUSIONS

While the MIMSY project involved relatively few people working on a project for a short time, it did confirm the usefulness of formal mathematical methods to the design of a system. MIMSY's "customers" judged the designs as being much more readable than the typical acquisition design document. All of the reviewers felt that they were able to grasp the functionality as well as the behavior that was in MIMSY's design.

The object orientation of the design methodology promoted information hiding, reusability and other concepts of software engineering. The notations provided by Ada, ANNA and Buhr diagrams all supported an object orientation. The Buhr diagrams provided a high level graphical representation of what was defined in detail in the textual Ada, ANNA and TSL.

Formal methods need to be applied to a system's behavior as well as its functionality. Ada's execution model provides a foundation for describing the behavior of a system. ANNA and TSL allow additional aspects of system behavior (beyond what Ada provides) to be formalized.

The use of Ada's (tasking) model supports executable designs and prototypes. There is no need to step outside of the PDL used with MIMSY in order to create an executable design or prototype. The design documents and the prototypes remain compatible.

Real-time embedded systems must be very robust, able to respond to a wide variety of inputs and error conditions. The rigor of ANNA encouraged designers and reviewers to cover more of these special cases in the design.

Building a design in hierarchies requires that all levels of the design have their functionality and behavior well-defined. The higher levels of the design will be created according to assumptions of the functionality and behavior of the lower levels. ANNA and TSL provide notations that can precisely define component behavior. If components are to be reused in different designs or systems, then their functionality and behavior must be precisely defined or else that component will not be very reusable.

Ada components, which are expected to be extensively reused, are especially vulnerable to different interpretations of system

behavior. The CAIS is supposed to be extensively used, yet the behavior of a CAIS implementation is subject to a variety of interpretations. ANNA's expressiveness points out the lack of behavioral requirements in existing specifications such as the CAIS.

Designers and reviewers need training in design methodology and notations of formal methods. Formal methods become useful only when all the audiences of the design can understand them. A design notation should not become an obstacle to understanding the system's design.

Formal methods are not part of the state of the practice. Experience in using formal methods on real systems is needed before formal methods become commonplace. The designers of real-time system have typically not had the opportunity to use formal methods in large real time embedded systems. The complexity of these systems also gives their designers the greatest need for formal methods.

Designers need reusable design paradigms (perhaps from earlier experiences) which can be used with ANNA and TSL. Otherwise the designers will have to start from scratch when formal methods are introduced.

Good ANNA and TSL tools are needed to use ANNA on real work. Trying to use formal methods by hand on large projects can be very tedious and expensive. A support environment is needed to make these formal methods cost effective on large projects.

MIMSY's use of the modified "lsedit" editor made using ANNA easier. This relieved the designers from having to remember all the details of ANNA's syntax.

The use of formal methods is only one part of the activities involved in creating a large scale software system. The other activities will have their own methodologies and support environments. ANNA tools must be part of an overall Ada programming and lifecycle support environment. Formal methods should not be treated in isolation but should instead be an integral part of the design. This means configuration management, change control and the other project activities called for by DOD-STD-2167.

ANNA and TSL do not address the design management information called for in DOD-STD-2167. This information is easier to collect and manipulate if it is kept with the design documents (such as PDL) that it describes. Users have to tailor ANNA to capture design management information (change logs, requirements traceability).

APPENDIX A  
ANNA EXAMPLE

```

with SCREEN_CONTROLLER, PROGRAM_CONTROLLER; package
STRUCTURE_MANAGER is

    MAXIMUM_NUMBER_OF_VIEWPORTS : NATURAL;
    MAXIMUM_NUMBER_OF_PROCESSES : NATURAL;

type COMMAND_RECORDS is record
    PROCESS_NAME : PROCESS_NAMES;
    VIEWPORT_NAME : VIEWPORT_NAMES;
    COMMAND_TYPE : COMMAND_TYPES;
end record;

procedure START;
--| where
--| preconditions
--| in MAXIMUM_NUMBER_OF_VIEWPORTS'DEFINED = FALSE and
--| in MAXIMUM_NUMBER_OF_PROCESSES'DEFINED = FALSE,

--| postconditions
--| out MAXIMUM_NUMBER_OF_VIEWPORTS =
--|     (TERMINAL_CAP.NUMBER_OF_LINES-2)/2,
--| out MAXIMUM_NUMBER_OF_PROCESSES =
--|     (TERMINAL_CAP.NUMBER_OF_LINES-2)/2;

procedure ANALYZE_COMMAND(COMMAND_RECORD: in COMMAND_RECORDS);
--| where
--| preconditions
--| in MAXIMUM_NUMBER_OF_VIEWPORTS'DEFINED = TRUE and
--| in MAXIMUM_NUMBER_OF_PROCESSES'DEFINED = TRUE,

procedure TERMINATE_ALL;
--| where
--| out SCREEN_CONTROLLER.NUMBER_OF_VIEWPORTS = 0 and
--|     PROGRAM_CONTROLLER.NUMBER_OF_PROCESSES = 0;

```

Figure 1. ANNA Example: Preconditions and Postconditions

```

--| axiom
--| for all SM : STRUCTURE_MANAGER`TYPE; C : COMMAND_RECORDS =>
--|   STRUCTURE_MANAGER`INITIAL.MAXIMUM_NUMBER_OF_VIEWPORTS`DEFINED
--|     = FALSE,
--|   STRUCTURE_MANAGER`INITIAL.MAXIMUM_NUMBER_OF_PROCESSES`DEFINED
--|     = FALSE,
--|   SM[ANALYZE_COMMAND(C)].MAXIMUM_NUMBER_OF_VIEWPORTS =
--|     SM.MAXIMUM_NUMBER_OF_VIEWPORTS,
--|   SM[ANALYZE_COMMAND(C)].MAXIMUM_NUMBER_OF_PROCESSES =
--|     SM.MAXIMUM_NUMBER_OF_PROCESSES,
--|   SM[TERMINATE_ALL].SCREEN_CONTROLLER.NUMBER_OF_VIEWPORTS = 0,
--|   SM[TERMINATE_ALL].PROGRAM_CONTROLLER.NUMBER_OF_PROCESSES = 0;
end STRUCTURE_MANAGER;

```

Figure 2. ANNA Example: Package Axioms

APPENDIX B  
CAISETTE EXAMPLE

package CAISETTE is

```
package CAIS_SCROLL_TERMINAL is
--/
--/ DESCRIPTION:
--/ This package encapsulates all the functions necessary to
--/ control a CRT scrolling terminal such as a VT100-class
--/ terminal. The terminal must be capable of supporting
--/ direct cursor addressing and be able to output bold
--/ (highlighted) as well as normal characters.

    procedure OPEN (TERMINAL : in FILE_TYPE ;
                    NODE      : out NODE_TYPE;
                    MODE      : in FILE_MODE := INOUT_FILE);

    -- raise STATUS_ERROR if the terminal is already open
    -- will open the terminal and pass the NODE back to caller

    procedure CLOSE (NODE : in NODE_TYPE);

    -- raise STATUS_ERROR if NODE isn't an open terminal
    -- will close terminal, preventing any further output
    -- attempts

    procedure SET_POSITION (NODE      : in NODE_TYPE;
                           POSITION : in POSITION_TYPE);

    -- NODE must be a currently open terminal
    -- will move the terminal's cursor to the position given
```

Figure 3. CAISette Without ANNA or TSL (OPEN and CLOSE)

```

procedure PUT (NODE : in NODE_TYPE;
               ITEM : in CHARACTER);

-- puts a character on terminal's screen at current
-- cursor position.  NODE must be an open terminal.

procedure PUT (NODE : in NODE_TYPE;
               ITEM : in STRING);

-- puts a string on screen at current cursor position.
-- NODE must be an open terminal.

end CAIS_SCROLL_TERMINAL;

private
type NODE_TYPE is new TBD; end CAISETTE;

```

Figure 4. CAISette Without ANNA or TSL (PUT)

package CAISETTE is

```
package CAIS_SCROLL_TERMINAL is
--/
--/ DESCRIPTION:
--/ This package encapsulates all the functions necessary to
--/ control a CRT scrolling terminal such as a VT100-class
--/ terminal. The terminal must be capable of supporting
--/ direct cursor addressing and be able to output bold
--/ (highlighted) as well as normal characters.

    procedure OPEN (TERMINAL : in FILE_TYPE ;
                    NODE      : out NODE_TYPE;
                    MODE      : in FILE_MODE := INOUT_FILE);

    --| where
    --| preconditions
    --| TERMINAL EXISTS(TERMINAL) /= TRUE =>
    --|   raise INTENTION_VIOLATION,
    --| TERMINAL NOW OPEN(TERMINAL) /= TRUE =>
    --|   raise STATUS_ERROR,

    --| exceptions will not affect package state
    --| raise INTENTION_VIOLATION =>
    --|   CAIS_SCROLL_TERMINAL = in CAIS_SCROLL_TERMINAL,
    --| raise STATUS_ERROR      =>
    --|   CAIS_SCROLL_TERMINAL = in CAIS_SCROLL_TERMINAL,

    --| postconditions
    --| out TERMINAL NOW OPEN(TERMINAL) = TRUE,
    --| out CURRENT_NODE(TERMINAL)      = NODE;
```

Figure 5. CAISette Example With ANNA (OPEN)

```

procedure CLOSE (NODE : in NODE_TYPE);
  --| where
  --| preconditions
  --| not exist T : FILE TYPE => CURRENT_NODE(T) = NODE and
  --|   TERMINAL NOW OPEN(T) = TRUE => raise STATUS_ERROR,
  --| raise STATUS_ERROR =>
  --| CAIS_SCROLL_TERMINAL = in CAIS_SCROLL_TERMINAL,

  --| postconditions
  --| out TERMINAL NOW OPEN(T) = FALSE,
  --| out CURRENT_POSITION      = (ROW => 1, COLUMN => 1);

  --| axiom
  --| for all P : CAIS_SCROLL_TERMINAL' TYPE; S,T : FILE_TYPE;
  --| N,M : NODE_TYPE; D : OPEN_DISPLAY =>

  --| basis step, initially no terminals are open

  --| CAIS_SCROLL_TERMINAL' INITIAL.TERMINAL_NOW_OPEN(T) = FALSE,

  --| induction step, if terminal last mentioned in OPEN then the
  --| terminal is still open, if last mentioned in CLOSE then
  --| the terminal still closed.

  --| P[OPEN(S,N)].TERMINAL_NOW_OPEN(T) =
  --| if S = T then TRUE else P.TERMINAL_NOW_OPEN(T),
  --| P[OPEN(S,N)].CURRENT_NODE(T) =
  --| if S = T then N else P.CURRENT_NODE(T),
  --| P[CLOSE(N)].TERMINAL_NOW_OPEN(T) =
  --| if exist U : FILE TYPE => CURRENT_NODE(U) = M and N = M
  --| and U = T then FALSE else P.TERMINAL_NOW_OPEN(T),
  --| P[CLOSE(N)].CURRENT_POSITION = (ROW => 1, COLUMN => 1),

```

Figure 6. CAISette Example With ANNA (CLOSE)

```

procedure SET_POSITION (NODE      : in NODE_TYPE;
                       POSITION : in POSITION_TYPE);
--
-- where -- preconditions
-- not exist T : FILE_TYPE => CURRENT_NODE(T) = NODE and
-- TERMINAL_NOW_OPEN(T) = TRUE => raise STATUS_ERROR,
-- POSITION.ROW > SCREEN_SIZE.ROW or POSITION.COLUMN >
-- SCREEN_SIZE.COLUMN => raise LAYOUT_ERROR, -- postconditions
-- out CURRENT_POSITION.ROW = POSITION.ROW and
-- CURRENT_POSITION.COLUMN = POSITION.COLUMN;
--: function UPDATE_POSITION (POSITION : POSITION_TYPE;
--: ITEM : CHARACTER) return POSITION_TYPE;
-- where return P : POSITION_TYPE =>
-- if ITEM = ASCII.BS
-- then if POSITION.COLUMN > 1
-- then P.COLUMN = POSITION.COLUMN - 1
-- else P.COLUMN = POSITION.COLUMN
-- endif
-- P.ROW = POSITION.ROW
-- else if ITEM = ASCII.LF
-- then if POSITION.ROW < SCREEN_SIZE.ROW
-- then P.ROW = POSITION.ROW + 1
-- else P.ROW = POSITION.ROW
-- endif
-- P.COLUMN = POSITION.COLUMN
-- endif
-- else if ITEM = ASCII.CR
-- then P.COLUMN = 1
-- P.ROW = POSITION.ROW
-- endif
-- else if ITEM = ASCII.HT
-- then P.COLUMN = P.COLUMN + 8 - ((P.COLUMN - 1) mod 8)
-- P.ROW = POSITION.ROW
-- endif
-- else if ITEM >= ASCII.SP and ITEM <= ASCII.TILDA
-- then P.COLUMN = POSITION.COLUMN + 1
-- P.ROW = POSITION.ROW
-- endif
-- else P = POSITION
-- endif

```

Figure 7. CAISette Example With ANNA (SET\_POSITION)

```

procedure PUT (NODE : in NODE_TYPE;
               ITEM : in CHARACTER);
    --| where
    --| preconditions

    --| not exist T : FILE_TYPE => CURRENT_NODE(T) = NODE and
    --|   TERMINAL_NOW_OPEN(T) = TRUE => raise STATUS_ERROR,

    --| postconditions
    --| out CURRENT_POSITION =
    --|   UPDATE_POSITION(CURRENT_POSITION, ITEM);

    procedure PUT (NODE : in NODE_TYPE;
                   ITEM : in STRING);
        --| where
        --| preconditions

        --| not exist T : FILE_TYPE => CURRENT_NODE(T) = NODE and
        --|   TERMINAL_NOW_OPEN(T) = TRUE => raise STATUS_ERROR,

        --| postconditions (see axioms, uses recursion of PUT(char))

        --| axiom
        --| for all P : CAIS_SCROLL_TERMINAL_TYPE; N : NODE_TYPE;
        --|   C, D : CHARACTER; S : STRING =>

    -- strings are output as characters with head recursion

    --|   P[PUT(C & S)] = P[PUT(C); UPDATE_POSITION(CURRENT_POSITION);
    --|                     PUT(S)],
    --|   P[PUT(C & D)] = P[PUT(C); UPDATE_POSITION(CURRENT_POSITION);
    --|                     PUT(D); UPDATE_POSITION(CURRENT_POSITION)];

end CAIS_SCROLL_TERMINAL;

private
    type NODE_TYPE is new TBD; end CAISETTE;

```

Figure 8. CAISette Example With ANNA (PUT)

package CAISETTE is

```
package CAIS_SCROLL_TERMINAL is
--/
--/ DESCRIPTION:
--/ This package encapsulates all the functions necessary to
--/ control a CRT scrolling terminal such as a VT100-class
--/ terminal. The terminal must be capable of supporting
--/ direct cursor addressing and be able to output bold
--/ (highlighted) as well as normal characters.

--: task type DISPLAY_EMULATOR is
--:   entry PUT (ITEM : in CHARACTER);
--:   entry FINISHED;
--: end DISPLAY_EMULATOR;

--: type OPENED_DISPLAY is access DISPLAY_EMULATOR;

procedure OPEN (TERMINAL : in FILE_TYPE ;
               NODE      : out NODE_TYPE;
               MODE      : in FILE_MODE := INOUT_FILE);

--+ when OPEN activates ?E where E is of type
--+   DISPLAY_EMULATOR then ?E running
--+   before ?E accepts at PUT;

procedure CLOSE (NODE : in NODE_TYPE);

--+ when not ?P calling ?E at PUT
--+   where ?P is of type CAIS_SCROLL_TERMINAL.PUT
--+   and ?E is of type DISPLAY_EMULATOR
--+   then CLOSE calling ?E at FINISHED
--+   before ?E terminated;

procedure SET_POSITION (NODE      : in NODE_TYPE;
                      POSITION : in POSITION_TYPE);

--+ when ?E accepts SET_POSITION at PUT
--+   where E is of type DISPLAY_EMULATOR
--+   then POSITIONING STRING'LENGTH occurrences of
--+   (?E accepts PUT => ?E releases PUT) before ?E
--+   releases SET_POSITION;
```

Figure 9. CAISette Example With TSL (OPEN and CLOSE)

```

procedure PUT (NODE : in NODE TYPE;
               ITEM : in CHARACTER);

--+ when ?E accepts CAIS_SCROLL_TERMINAL.PUT at PUT
--+   where E is of type DISPLAY EMULATOR
--+   and where GRAPHIC_RENDERITION_SUPPORT = BOLD then
--+   BEGIN_BOLD'LENGTH + 1 + END_BOLD'LENGTH occurrences of
--+   (?E accepts PUT => ?E releases PUT) before
--+   ?E releases CAIS_SCROLL_TERMINAL.PUT;

procedure PUT (NODE : in NODE TYPE;
               ITEM : in STRING);

--+ when ?E accepts CAIS_SCROLL_TERMINAL.PUT at PUT
--+   where E is of type DISPLAY EMULATOR
--+   and where GRAPHIC_RENDERITION_SUPPORT=PRIMARY_RENDERITION
--+   then ITEM'LENGTH occurrences of (?E accepts PUT =>
--+   ?E releases PUT)
--+   before ?E releases CAIS_SCROLL_TERMINAL.PUT;

--+ when ?E accepts CAIS_SCROLL_TERMINAL.PUT at PUT
--+   where E is of type DISPLAY EMULATOR
--+   and where GRAPHIC_RENDERITION_SUPPORT = BOLD then
--+   BEGIN_BOLD'LENGTH + ITEM'LENGTH + END_BOLD'LENGTH
--+   occurrences of (?E accepts PUT => ?E releases PUT) before
--+   ?E releases CAIS_SCROLL_TERMINAL.PUT;
end CAIS_SCROLL_TERMINAL; end CAISETTE;

```

Figure 10. CAISette Example With TSL (PUT)

## REFERENCES

- Berg82    Berg, H. K., et al., Formal Methods of Program Verification and Specification, Prentice-Hall, 1982.
- Buhr84    Buhr, R.J.A., System Design With Ada, Prentice Hall, 1984.
- Cher85    Cherry, George, "The PAMELA Methodology, A Process-Oriented Software Development Method for Ada," Proceedings of SIGAda/AdaJUG Conference, Association of Computing Machinery, 1985.
- Cloc81    Clocksin, William and Mellish, Christopher, Programming in Prolog, Springer-Verlag, 1981.
- DEC85a    Digital Equipment Corporation, VAX Language-Sensitive Editor, AA-DB33A-TE, March, 1985.
- DEC85b    Digital Equipment Corporation, Developing Ada Programs on VAX/VMS, AA-EF86A-TE, February, 1985.
- DOD83    Department of Defense, Ada Joint Program Office, Ada Language Reference Manual, ANSI/MIL-STD-1815A-1983, 1983.
- DOD85    Department of Defense, Defense Software Development Standard, DOD-STD-2167, June, 1985.
- ESPR85    European Strategic Programme for Research in Information Technology (ESPRIT), PROgram Development by SPECification and TRANSformation (PROSPECTRA) Project Summary, ESPRIT Programme of the Commission of the European Communities, 1985.
- Gerh80    Gerhart, Susan, et al., An Overview of AFFRIM: A Specification and Verification System, PR-79-81, USC Information Sciences Institute, 1980.
- Gold83    Goldberg, Adele, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.
- Goos83    Goos, Gerhard and Wulf, William (editors), DIANA, An Intermediate Language for Ada, Lecture Notes in Computer Science 161, Springer-Verlag, 1983.

## REFERENCES (Concluded)

- Gutt78 Gutttag, John, Horowitz, Ellis and Musser, David, "Abstract Data Types and Software Validation," Communications of the ACM, Volume 21, Number 12, December 1978.
- Helm84 Helmbold, David and Luckham, David, Debugging Ada Tasking Programs, Technical Report 84-262, Stanford University, July 1984.
- Helm85 Helmbold, David and Luckham, David, "TSL: Task Sequencing Language," Ada in Use, Proceedings of the Ada International Conference, Volume V, Issue 2, Association of Computing Machinery, 1985.
- Inte84 The Byron User's Manual, Version 1.1, Intermetrics, Inc., 1984.
- KIT84 KIT/KITIA, Common APSE Interface Set (CAIS), Version 1.4, 1984.
- Luck84 Luckham, David, et al, ANNA, A Language For Annotating Ada Programs, TR-84-261, Computer Systems Laboratory, Stanford University, 1984.
- Stal85 Stallman, Richard, GNU Emacs Manual, Emacs Version 16, June 1985.
- TI83 APSE Interactive Monitor, Program Performance Specification, CDRL Seq. No. A001, Texas Instruments Incorporated, 1983.